BO-IRL Code Submission

This document contains instructions on how to repeat the experiments presented in our paper titled "Efficient Exploration of Reward Functions in Inverse Reinforcement Learning via Bayesian Optimization". BO-IRL is a novel IRL algorithm that uses Bayesian Optimization for efficient exploration of the reward space to identify multiple regions of high likelihood w.r.t to the expert demonstrations.

The actual code repository is uploaded in a zip file titled "BOIRLCode.zip". It contains the implementation of BO-IRL with the novel ρ-RBF kernel as well as standard RBF kernel and Matern Kernel.
We have also included code to test our results against
Guided Cost Learning(GCL), Adversarian IRL (AIRL) and Bayesian IRL (BIRL).

Four environments are available in this code (as mentioned in the paper):

- `Gridworld` : Synthetic gridworld experiment where the states are represented by the number of gold coins present in them.
- `Real Borlange` : Borlange dataset with real world trajectories and no ground truth reward function
- `Point Mass Maze` : A modified version of the mujoco environment introduced in Adversarian IRL (AIRL)
- `Fetch-Reach` : One of the Fetch simulations available in Open AI

In addition, we have two more environments that are variants of the Gridworld and Real Borlange environments mentioned above:

- `Gridworld2d` : A simplified Gridworld environment where the translation parameter (see paper) is removed.
- `Virtual Borlange` : Simulated environment corresponding to the Borlange dataset with an artificial reward function.

While running code, you can refer to these environments using the following `<envref>`

envref	Environment Name
grdiworld2d	Gridworld2d
grdiworld3d	Gridworld
vborlange	Virtual Borlange
rborlange	Real Borlange
maze	Point Mass Maze
fetch	Fetch-Reach

Download the project.

The project has the following requirements.

- `Python3`
- Standard Anaconda packages such as `Numpy, Scipy, Matplotlib`
- `Tensorflow (==1.14.0)` for AIRL, GCL
- `tqdm (==4.24.0)`
- `tabulate(==0.8.6)`

If you wish to analyze existing results, you will need to download the full sized project from it's Github repository. However, if you wish to train BO-IRL from scratch, you can unzip the attached "BOIRLCode.zip" into a local directory. In both cases, we will assume that the full path of the local directory is: '/home/user/BOIRLSubmission'. We will refer to this path as "basedir". We will also assume you have browsed to this location in your terminal.

Setting up OpenAI Gym

In our expertiments, we have modified the Fetch-Reach environment in OpenAI Gym to take in reward function parameters as inputs. This is needed when evaluating a particular reward function. Similarly, we have also added the Point-Mass Maze environment as a standard OpenAI environment. As a result, to run the codes in this package successfully, you will need to uninstall any standard OpenAI gym present in your python virtual environment (or create a new virtual/conda environment) and install it from our repository. You can follow the steps below:

```
cd gym_boirl
pip install -e .
cd ..
```

You will need to have Mujoco support for Open AI gym. You can follow the instructions from this page on how to setup mujoco for use with OpenAI Gym.

For GCL and IRL

We have used the implementation of GCL and AIRL found at this repository. Make sure that the necessary prerequisites are satisfied. This repository requires the environment to be an OpenAI Gym. Therefore, we have provided an OpenAI Gym version of Gridworld3d, Virtual Borlange and Real Borlange. To activate them, follow the steps below:

For Gridworld3d:

```
cd gym-sigmoid
pip install -e .
cd ..
```

For Virtual Borlange:

Open "swedenworld_env.py" inside gym-sweden/gym_sweden/envs in a text editor. Change the variable self.basedir to point to the location of your basedir using it's absolute path. Save the file

```
cd gym-sweden
pip install -e .
cd ..
```

Evaluating Existing Results Without Retraining

As mentioned above, please download the project from it's Github repository. This project folder contains the results used for Table 1 as well as Figure 5 and 6. We now discuss how to regenerate these results:

For Table 1:

Table 1 in our paper presents the success rate and number of iterations required for each algorithm to catch up to the expert's ESOR. For instance, our results for Gridworld environment looks as follows:

Algorithm	Kernel	SR	ESOR
BO-IRL	ρ-RBF	70%	16.0 +/- 15.6
BO-IRL	RBF	50%	30.0 +/- 34.4
BO-IRL	Matern	60%	22.2 +/- 12.2

Algorithm	Kernel	SR	ESOR
AIRL	-	70%	70.4 +/- 23.1
GCL	-	40%	277.52 +/- 113.1

To generate this table from existing results, run the following:

```
sh runTable1.sh
```

This will print the results corresponding to Table 1 (there might be slight differences in the values due to change in expert demonstrations between experiments). It will also plot the ESOR and NLL progress for the algorithms being evaluated and place them on the `basedir`. For Fetch-Reach environment, the success rate plot is generated instead of ESOR.

For Figure 5 and 6:

To generate the plots used in Figures 5 and 6, run the following:

```
python posterior_plot.py
```

This takes in selected results from `PosteriorPlotsFromPaper/Data` and plots the corresponding figures in `PosteriorPlotsFromPaper/Plots`.

Generating Expert Trajectory (Optional)

Before running any IRL algorithms, you need to collect expert trajectories (and other metadata) for the environment of interest. This step needs to be performed only once per environment as the same set of expert trajectories will be reused by all IRL algorithms. To collect expert trajectories on a given environment, run the following by replacing `<envref>` with the correct reference to the environment of interest (as mentioned in the table above). **This step is optional since the expert trajectories are provided both in the zip file and the Github repository. Also, this does not apply to Point Mass Maze and Fetch-Reach environments**

Warning: Do not delete the existing Data folder

```
python datacollect.py -e <envref>
```

The trajectories will be saved to the `Data` folder under the basedir.

IRL Algorithms

The following algorithms are available:

algoref	Algorithm Name
rhorbf	BO-IRL with ρ-RBF
rbf	BO-IRL with RBF
matern	BO-IRL with Matern

algoref	Algorithm Name
gcl	GCL
airl	AIRL
birl	BIRL

Compare Posterior Mean and Std.

Our paper compares the posterior mean and std generated using BO-IRL+ ρ-RBF kernel to the posterior distribution over reward functions learned by BIRL (Fig. 5). We also compare the GP posterior obtained by ρ-RBF kernel with those from standard kernels such as RBF and Matern (Fig. 9). To generate these results, we need to run BO-IRL algorithm as shown below:

```
python runalgorithm.py -e <envref> -a <algoref> -b <budget> -n <number of trials>
```

GP posterior mean and std can be visualized for environments whose reward function space has dimension=2. One exception to this rule is Virtual Borlange which plots the GP Posterior by setting the third dimention to -1. Therefore, valid `<envref>` values for visualizing the GP posterior include `gridworld2d`, `maze`, `fetch` and `vborlange`. `<algoref>` should be one of the BO-IRL methods, i.e. `rhorhf`, `rbf` or `matern`. `<budget>` is an integer that corresponds to the total budget (or total number of iterations) for optimization. By default, the BO is initialized with 1 additional sample at the beginning of the iteration (so total samples after optimization = budget + 1). `<number of trials>` is an integer that specifies how many times to run this experiment.

For instance, to generate the posterior mean and std for Gridworld2d using BO-IRL with ρ-RBF kernel with a budget of 30 iterations and repeated 3 times, we can run the following:

```
python runalgorithm.py -e gridworld2d -a rhorbf -b 2 -n 3
```

Once the code is executed, plots of the posterior mean and std can be found in `<basedir>/ResultDir/<envref>/<algoref>/` under the names `PosteriorMean<trial>.png` and `PosteriorStd<trial>.png` respectively.

As shown in the paper (Fig. 5), we can compare the results of the discrete environments, namely Gridworld2d and Virtual Borlange against BIRL. To do so, run the following commands:

For Gridworld2d:

```
python birl_gridworld.py
```

For Virtual Borlange:

```
python birl_vborlange.py
```

The results will be placed in `<basedir>/ResultDir/<envref>/birl/`. **However, we since this evaluation can be time consuming, we have already placed our previous results in the corresponding folders for your inspection without having to run the code.**

Number of iterations to reach expert's ESOR.

Our paper compares BO-IRL (with ρ-RBF, RBF and Matern kernels) to GCL and AIRL in it's ability to reach expert's Expected Sum of Rewards (ESOR) faster in all environments except Fetch-Reach and Real Borlange. The following section explains how to generate these results. In addition to reporting the number of iterations, we also plot the ESOR and NLL for each algorithm across the training iterations. Please note that ESOR calculation is not valid for Real Borlange as we do not have access to a ground truth reward function. So running the following codes will only result in the NLL plot.

1. Run the corresponding IRL algorithms for the given environment for a specified budget.

```
python runalgorithm.py -e <envref> -a <algoref> -b <budget> -n <number of trials>
```

Here `<algoref>` can be any algorithm except `birl`. Run this code once for each algorithm that you want to examine.

2.Calculate the ESOR and NLL using the output of the specified algorithm across iterations. (Warning: This might be time consuming for environments other than the gridworld).

```
python evaluatealgorithm.py -e <envref> -a <algoref>
```

Run this code once for each algorithm that you want to examine.

3.Plot the results

```
python plotalgorithm.py -e <envref> -a <algoref> <algoref> <algoref>..  -p <Percentage of expert's ESOR to ma
```

When calling this script, you can pass multiple `<algoref>` separated by space. `<Percentage of expert's ESOR to match>` is a value between 0-1 that represents what percentage of expert's ESOR the algorithm should aim to reach (defaults to 1). `<Number of iterations to plot>` is an integer that represents number of iterations to plot ESOR and NLL against.

Results:

- ESOR plot will be saved in `basedir` as `ESOR_<envref>.png`
- NLL plot will be saved in `basedir` as `NLL_<envref>.png`
- Number of iterations for each algorithm (mean and std) to reach the given percentage of Expert's ESOR will be printed in console.

Success Rate for Fetch-Reach

Instead of calculating the ESOR for each iteration, we evaluate the performance of BO-IRL in the Fetch-Reach environment using the success rate (SR) metric. Success rate indicates the number of times the robot's gripper touched the target in a given number of episodes. We compare the success rate of the agent trained using the learned reward function using various kernels. AIRL and GCL were not tested due to incompatabilities with the libraries.

The steps involved are identical to the 3 steps mentioned above (`envref` is set to `fetch`). However, instead of generating the ESOR plot, a SR plot will be placed in `basedir` as `SR_fetch.png`